Undefined behaviour in the Standard Template Library

Sandor DARGO

16th June 2020

C++ On Sea

Who Am I?

Sándor DARGÓ

- Software developer in Amadeus
- Enthusiastic blogger http://sandordargo.com
- Passionate traveller
- Curious home baker
- Happy father of two

Agenda

Vocabulary

Reasons of undefined behaviour (in the STL)

Types of UB in the STL

Countermeasures

Let’s agree on a common vocabulary!

Let there be a generic library!

Alexander Stepanov had a dream

1970s: Had the idea in mind,
but no language support

1987: First try in Ada

1993 Nov: First presentation to the Committee

1994 March: Formal proposal

1994 July: Final approval of the Committee

1994 August: HP publishes first implementation

What is included in the STL?

There is no such “library” as the STL

Set of template classes and functions to provide solution for common problems

- Algorithms (std::rotate, std::find_if)
- Containers (std::vector<T>, std::list<T>)
- Function objects (std::greater<T>, std::logical_and<T>)
- Iterators (std::iterator, std::back_inserter)

What is (observable) behaviour of code?

Guaranteed behaviour

Ill-formed

Ill-formed no diagnostic required

Unspecified behaviour

Implementation defined behaviour

Undefined behaviour

Unspecified behaviour

Rules are not specified by the Standard

Implementation doesn't have to document

Different result sets are valid

No crash, strictly limited perimeters

Examples:

- &x > &y
- expression evaluation order

```
24  #include <iostream>
25  int x=333;
26
27  int add(int i, int j) {return i+j;}
28  int left() {
29    x = 100;
30    return x;
31  }
32  int right() {
33    x++;
34    return x;
35  }
36  int main() {
37   std::cout << add(left(), right()) << std::endl;
38  }
39
40
```

```
434
```

```
g++ -std=c++14 -Wall -pedantic -pthread mai
```

```
clang version 7.0.0-3~ubuntu0.18.04.1 (tags/RELEASE_700/final)
> clang++-7 -pthread -o main main.cpp
> ./main
201
>
```

Implementation defined behaviour

Like unspecified behaviour

But implementation must document the result

Examples:

- Default integer types
- Number of bits in a byte
- ORDER BY on NULLs

Undefined behaviour

We break the rules, no requirements on the behaviour

Anything can happen to the entire program, the compiler owes us nothing

- Crash
- Logically impossible results
- Non-deterministic behaviour
- Removed execution paths

Examples

- Accessing uninitialized variables
- Deleting object through base class pointer w/o virtual destructor

Why does UB exist?

Portability

Performance optimizations

Make APIs shorter

Simpler implementations

Why UB is allowed in the STL?

- Efficiency
- Shorter API
- Implementation freedom

Types of UBs in the STL

Pointing beyond the limits of a container

Inconsistent iterators

Unsorted data

And the others

Beyond the limits

operator[] vs at()

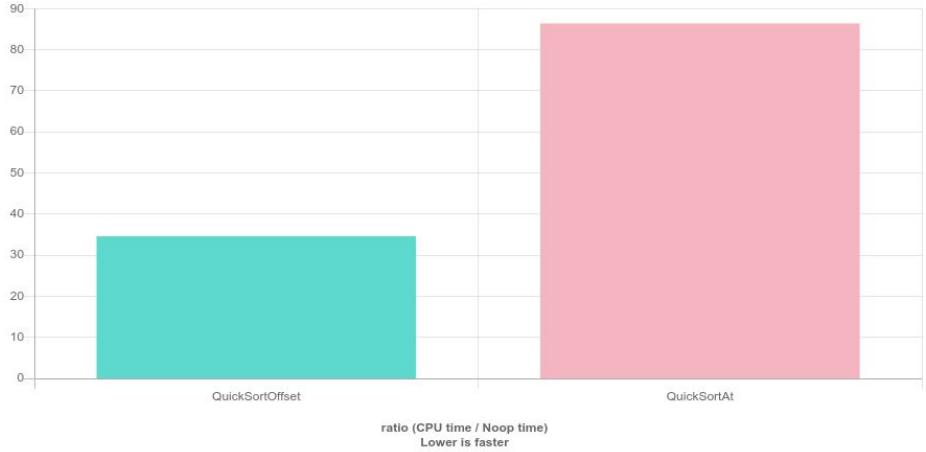
Pay for what you use

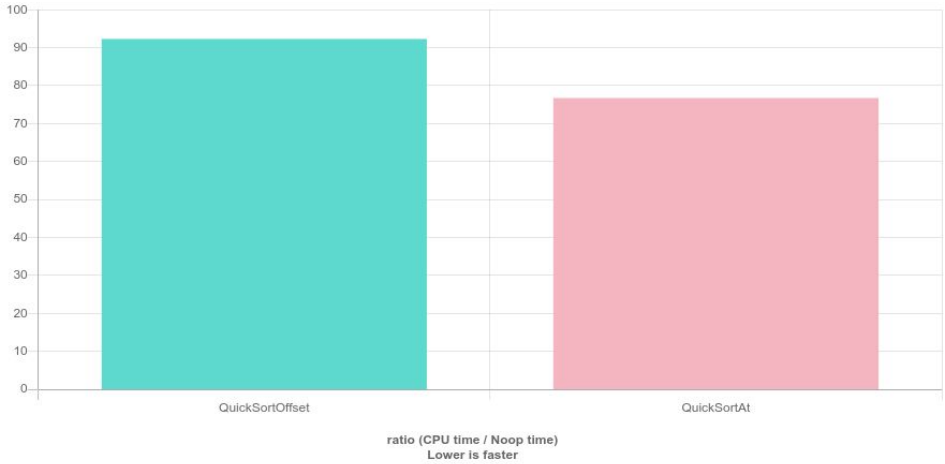
No bound checks vs throwing exception

Potentially significant performance overhead (250%) if not optimized

Both O(1) Complexity

Don’t rely on optimizations, choose what you need, but *at()* is rarely needed





front()/back() et al.

Getting an item from an empty container is UB

- back()
- front()
- pop_back()
- pop_front()

Possible outcomes:

- Segmentation fault
- Double free
- Random value

erase() invalidates iterators

Don’t modify a container while iterating over it

erase() invalidates positions

What happens after is undefined:

- Nothing observable
- Unexpected result (value not removed)
- Segmentation fault

```
std::vector<int> numbers{1,2,3,4,5,6,4};

int val = 4;
for (auto it = numbers.begin();
          it != numbers.end(); ++it) {
    if (*it == val) {
        numbers.erase(it);
    }
}

std::cout << "numbers after erase:";
for (const auto num : numbers)
    std::cout << num << " ";
```

Inconsistent iterators

Ranges are defined by one or two iterators

```
template< class ExecutionPolicy, class ForwardIt 1, class
ForwardIt2, class ForwardIt3, class BinaryOperation >
ForwardIt3 transform( ExecutionPolicy&& policy,
                      ForwardIt1 first1, ForwardIt1 last1,
                      ForwardIt2 first2, ForwardIt3 d_first,
                      BinaryOperation binary_op );
```

first1, last1 - the first range of elements to transform

first2 - the beginning of the second range of elements to transform

d_first - the beginning of the destination range, may be equal to first1 or first2

Algorithms don’t validate ranges

Ranges might be combined

But don’t trust it…

```cpp
#include <algorithm>
#include <iostream>
#include <vector>

int main() {
  auto numbers21 = { 1, 3 };
  auto numbers22 = { 3, 5 };
  std::vector<int> copiedNumbers;

  std::copy_if(numbers21.begin(), numbers22.end(),
    std::back_inserter(copiedNumbers),
    [](auto number) {return number % 2 == 1;});

  std::cout << " copied numbers: ";
  for (const auto number : copiedNumbers) {
    std::cout << ' ' << number;
  }
  std::cout << '\n';

  return 0;
}
```

```
copied numbers:  1 3 3 5
```

Algorithms don’t validate ranges

Ranges might be combined

But don’t trust it…

Different types work differently

```cpp
#include <algorithm>
#include <iostream>
#include <vector>

int main() {
  std::vector<int> numbers21{ 1, 3 };
  std::vector<int> numbers22{ 3, 5 };
  std::vector<int> copiedNumbers;

  std::copy_if(numbers21.begin(), numbers22.end(),
    std::back_inserter(copiedNumbers),
    [](auto number) {return number % 2 == 1;});

  std::cout << " copied numbers: ";
  for (const auto number : copiedNumbers) {
    std::cout << ' ' << number;
  }
  std::cout << '\n';

  return 0;
}
```

```
copied numbers:  1 3 33 3 5
```

Algorithms don’t validate ranges

Ranges might be combined

But don’t trust it…

Different types work differently

And you might get very strange results

```cpp
#include <algorithm>
#include <iostream>
#include <list>
#include <vector>

int main() {
  std::list<int> numbers21{ 1, 3 };
  std::list<int> numbers22{ 3, 5 };
  std::vector<int> copiedNumbers;

  std::copy_if(numbers21.begin(), numbers22.end(),
    std::back_inserter(copiedNumbers),
    [](auto number) {return number % 2 == 1;});

  std::cout << " copied numbers: ";
  for (const auto number : copiedNumbers) {
    std::cout << ' ' << number;
  }
  std::cout << '\n';

  return 0;
}
```

```
execution expired
```

Algorithms validate types

At least container types cannot be combined

```
1  #include <algorithm>
2  #include <iostream>
3  #include <vector>
4  #include <list>
5
6  int main() {
7    std::vector<int> numbers21{ 1, 3 };
8    std::list<int> numbers22 = { 3, 5 };
9    std::vector<int> copiedNumbers;
10
11   std::copy_if(numbers21.begin(), numbers22.end(),
12     std::back_inserter(copiedNumbers),
13     [](auto number) {return number % 2 == 1;});
14
15   std::cout << " copied numbers: ";
16   for (const auto number : copiedNumbers) {
17     std::cout << ' ' << number;
18   }
19   std::cout << '\n';
20
21   return 0;
22 }
```

```
main.cpp: In function 'int main()':
main.cpp:13:46: error: no matching function for call to
   13 |     [](auto number) {return number % 2 == 1;});
      |                                              ^
      |
```

Algorithms validate types

Nor the contained types!

```cpp
#include <algorithm>
#include <iostream>
#include <vector>

int main() {
  std::vector<int> numbers21{ 1, 3 };
  std::vector<unsigned int> numbers22 = { 3, 5 };
  std::vector<int> copiedNumbers;

  std::copy_if(numbers21.begin(), numbers22.end(),
    std::back_inserter(copiedNumbers),
    [](auto number) {return number % 2 == 1;});

  std::cout << " copied numbers: ";
  for (const auto number : copiedNumbers) {
    std::cout << ' ' << number;
  }
  std::cout << '\n';

  return 0;
}
```

```
main.cpp: In function 'int main()':
main.cpp:12:46: error: no matching function for call to
   12 |     [](auto number) {return number % 2 == 1;});
      |                                              ^
      |
```

Types cannot be combined

Unless...

We inject in the middle a different contained type

```cpp
#include <algorithm>
#include <iostream>
#include <vector>

int main() {
  std::vector<int> numbers21{ 1, 3 };
  std::vector<float> different = { 3.14, 4.2 };
  std::vector<int> numbers22 = { 3, 5 };
  std::vector<int> copiedNumbers;

  std::copy_if(numbers21.begin(), numbers22.end(),
    std::back_inserter(copiedNumbers),
    [](auto number) {return number % 2 == 1;});

  std::cout << " copied numbers: ";
  for (const auto number : copiedNumbers) {
    std::cout << ' ' << number;
  }
  std::cout << '\n';

  return 0;
}
```

```
copied numbers:  1 3 33 1078523331 33 3 5
```

Types cannot be combined

Or even a different container type

```cpp
#include <algorithm>
#include <iostream>
#include <vector>
#include <list>

int main() {
  std::vector<int> numbers21{ 1, 3 };
  std::list<float> different = { 3.14 };
  std::vector<int> numbers22 = { 3, 5 };
  std::vector<int> copiedNumbers;

  std::copy_if(numbers21.begin(), numbers22.end(),
    std::back_inserter(copiedNumbers),
    [](auto number) {return number % 2 == 1;});

  std::cout << " copied numbers: ";
  for (const auto number : copiedNumbers) {
    std::cout << ' ' << number;
  }
  std::cout << '\n';

  return 0;
}
```

```
copied numbers:  1 3 33 32765 32765 1078523331 33 3 5
```

Size matters

We shall respect contracts

Or we go to random places

```cpp
#include <iostream>
#include <algorithm>
#include <vector>

int main () {
auto values = std::vector<int>{1,2,3,4,5,6,7,8,9};
auto otherValues = std::vector<int>{10,20,30};
auto results = std::vector<int>{};
std::transform(values.begin(), values.end(),
    otherValues.begin(),
    std::back_inserter(results),
    [](int number, int otherNumber) {return number+otherNumber;});
std::for_each(results.begin(), results.end(),
    [](int number){ std::cout << number << "\n";});
return 0;
}
```

```
11
22
33
4
5
6
40
8
31304841
```

Size matters

Or breaking the contract can even lead to a core dump

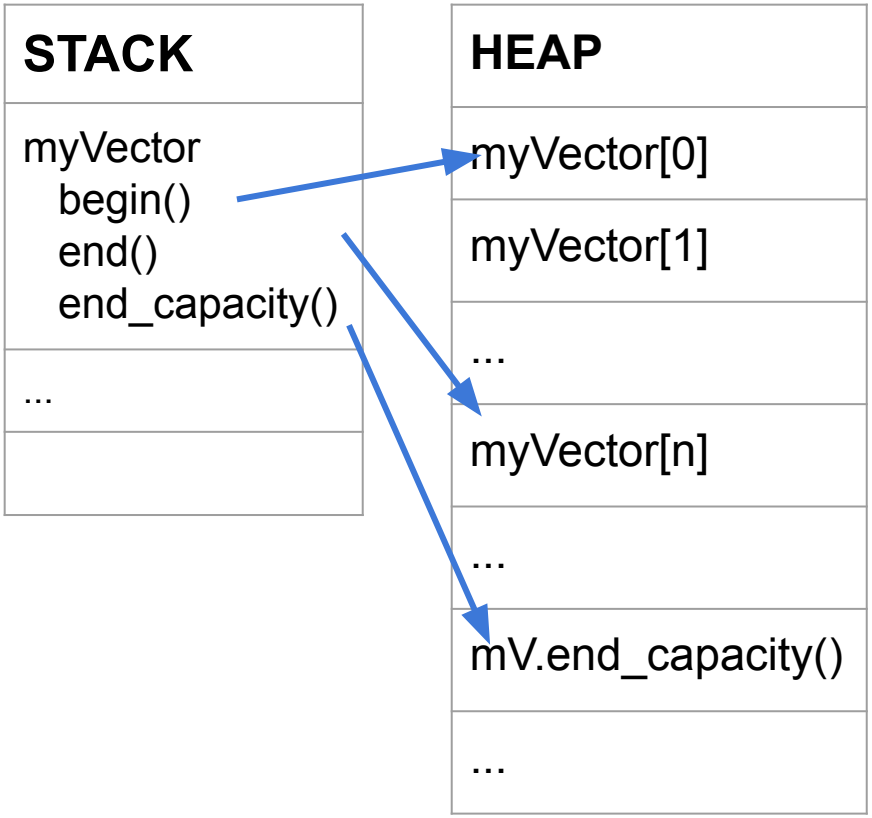
```
5  struct I{
6    int number;
7  };
8
9  int main () {
10 auto values = std::vector<I*>{new I{1},new I{2},new I{3},new I{4},new I{5}};
11 auto otherValues = std::vector<I*>{new I{10},new I{20},new I{30}};
12 auto resutls = std::vector<int>{};
13 std::transform(values.begin(), values.end(),
14     otherValues.begin(), std::back_inserter(resutls),
15     [](I* n, I* m) {
16         std::cout << "number: " << n->number << std::endl;
17         std::cout << "another number: " << m->number << std::endl;
18         return n->number + m->number;
19     });
20
21 std::for_each(resutls.begin(), resutls.end(),
22     [](int number){ std::cout << number << "\n";});
23 return 0;
24 }
```

```
number: 1
another number: 10
number: 2
another number: 20
number: 3
another number: 30
number: 4
bash: line 7: 13019 Segmentation fault      (core dumped) ./a.out
```

What's going on?

Container overhead is on the stack and data is on the heap

Iterators are just advanced by one



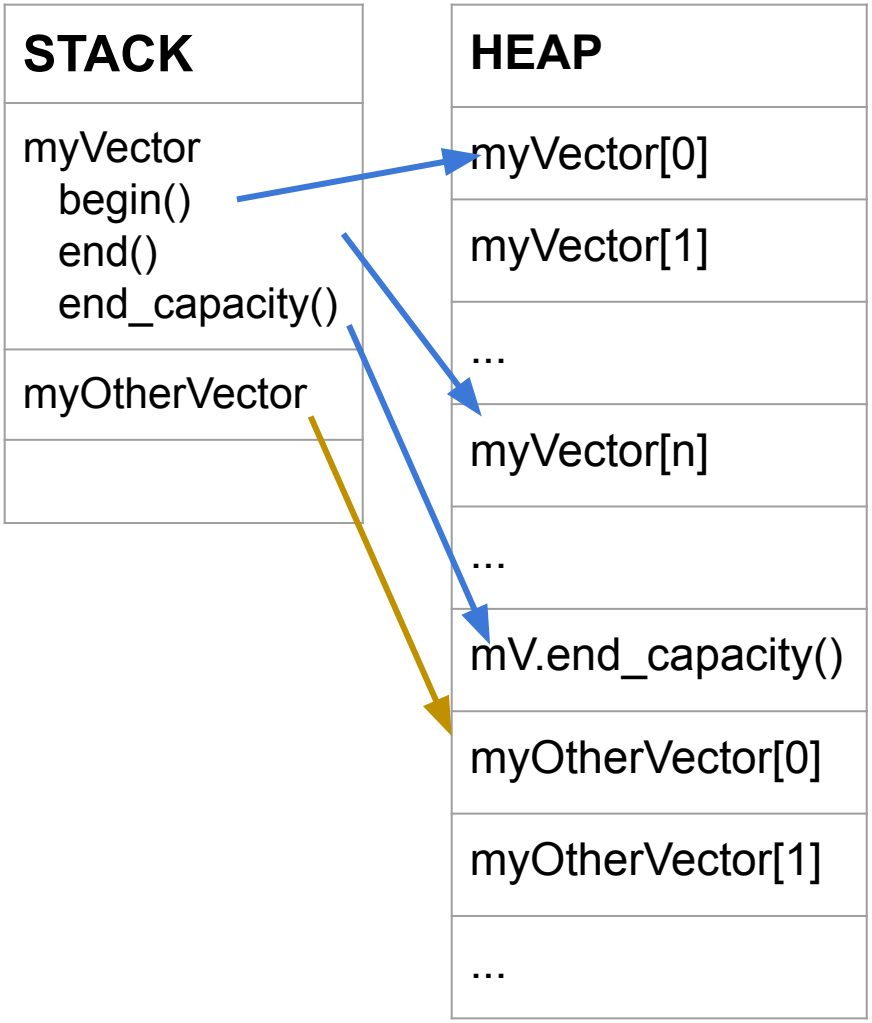
What's going on?

Container overhead is on the stack and data is on the heap

Iterators are just advanced by one

Sometimes to the next container

Sometimes to uncharted territories



Algorithms don’t validate ranges

No explicit error of mixed ranges

But you might get

- Compilation error
- Runtime error
- Strange results

After all, this is true undefined behaviour

Double check what you pass in…

Unsorted data

Will it find it?

SORTED-RANGE ALGORITHMS for sequences that are not sorted on entry results in undefined behavior.

```cpp
#include <algorithm>
#include <iostream>
#include <vector>

int main() {
    std::vector<int> numbers{1,54,7,5335,8};
    std::cout << " 7 is found at position: "
              << std::binary_search(numbers.begin(), numbers.end(), 7)
              << std::endl;
    std::sort(numbers.begin(), numbers.end());
    std::cout << " 7 is found at position: "
              << std::binary_search(numbers.begin(), numbers.end(), 7)
              << std::endl;
}
```

```
7 is found at position: 0
7 is found at position: 1
```

Algorithms come with their contracts!

Will it find it?

`lower_bound()` and `upper_bound()` might also return you incorrect data if source is unsorted

```cpp
#include <algorithm>
#include <iostream>
#include <vector>

int main() {

    std::vector<int> numbers{1, 888, 54, 7, 5335, 7, 8,3, 7};
    auto lower = std::lower_bound(numbers.begin(), numbers.end(), 7);
    std::cout << " 7 is found at position "
              << (lower - numbers.begin()) << std::endl;
    std::cout << " But what is there? "
              << *lower << std::endl;
    std::sort(numbers.begin(), numbers.end());
    lower = std::lower_bound(numbers.begin(), numbers.end(), 7);
    std::cout << " After sorting 7 is found at position "
              << (lower - numbers.begin()) << std::endl;
    std::cout << " But what is there now? "
              << *lower << std::endl;
}
```

```
7 is found at position 1
But what is there? 888
After sorting 7 is found at position 2
But what is there now? 7
```

Do you understand merging?

Merge will most likely work and will give a not completely meaningless result

But it also expects sorted containers

```cpp
#include <algorithm>
#include <iostream>
#include <vector>

int main() {

    std::vector<int> oddNumbers{1,9,3,7,5};
    // std::sort(oddNumbers.begin(), oddNumbers.end());
    std::vector<int> evenNumbers{2,10,4,8,6};
    // std::sort(evenNumbers.begin(), evenNumbers.end());
    std::vector<int> mergedNumbers;
    std::merge(oddNumbers.begin(), oddNumbers.end(),
        evenNumbers.begin(), evenNumbers.end(),
        std::back_inserter(mergedNumbers));

    std::cout << "The merged numbers are:";
    for (auto number : mergedNumbers) {
        std::cout << number << " ";
    }
    std::cout << std::endl;
}
```

```
The merged numbers are:1 2 9 3 7 5 10 4 8 6
```

Why not merging them?

Merge will most likely work as one would expect - or in a similar way -, but we cannot rely on it

For this output you don't need merge.

```
 6
 7  int main() {
 8
 9      std::vector<int> oddNumbers{1,9,3,7,5};
10      // std::sort(oddNumbers.begin(), oddNumbers.end());
11      std::vector<int> evenNumbers{2,10,4,8,6};
12      // std::sort(evenNumbers.begin(), evenNumbers.end());
13      std::vector<int> mergedNumbers;
14      std::merge(oddNumbers.begin(), oddNumbers.end(),
15          evenNumbers.begin(), evenNumbers.end(),
16          std::back_inserter(mergedNumbers),
17          std::greater<int>());
18
19      std::cout << "The merged numbers are:";
20      for (auto number : mergedNumbers) {
21          std::cout << number << " ";
22      }
23      std::cout << std::endl;
24  }
25
26
27
```

```
The merged numbers are:2 10 4 8 6 1 9 3 7 5
```

Others

Pay attention when splicing

This and source must be different

Lot of relative rules

Many combinations can go wrong

Prepare for the bug, test & wrap

```cpp
#include <iostream>
#include <list>

int main() {
  std::list<int> odds{1, 9, 3, 5, 7};
  std::list<int> evens{2, 10, 4, 8, 6};
  odds.pop_front();
  odds.splice(odds.begin(), evens, odds.begin(), odds.end());
  for (auto n: odds) {
    std::cout << " " << n;
  }
  std::cout << std::endl;
  return 0;
}
```

```
9 3 5 7 9 3 5 7 9 3 5 7 9 3 5 7 9 3 5 7 9 3 5 7 9 3 5
```

```cpp
#include <iostream>
#include <list>

int main() {

  std::list<int> odds{1, 9, 3, 5, 7};
  std::list<int> evens{2, 10, 4, 8, 6};
  auto it = odds.begin();
  odds.pop_front();
  odds.splice(it, evens);
  return 0;
}
```

```
bash: line 7:  9747 Segmentation fault
```

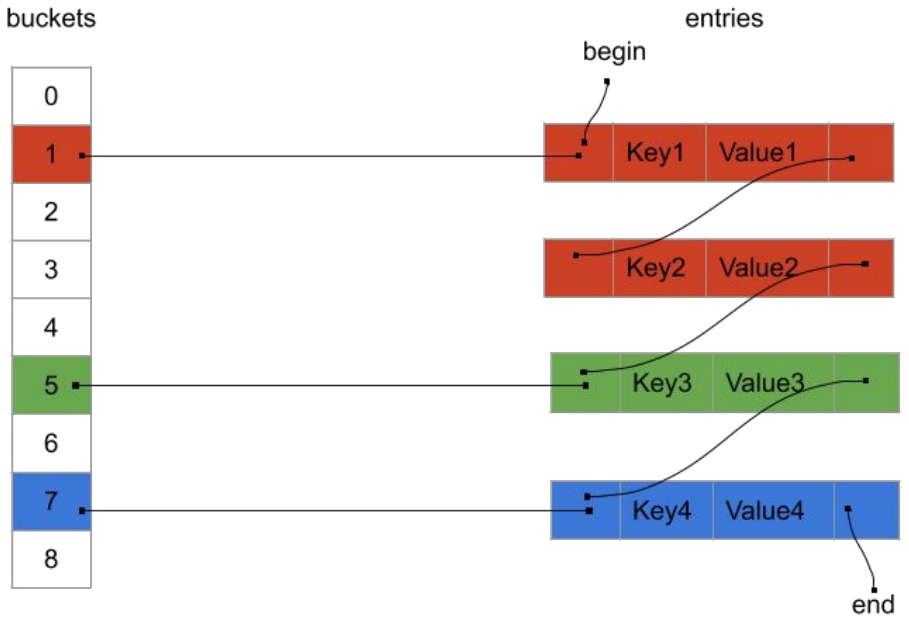
unordered_map also comes with UBs

`size_type container::bucket (const key_type key) const`

Returns the index of the bucket for a given key

The return value is undefined if bucket_count() is zero.

In practice it doesn't happen as UMs are initialized with some size



bucket_size is similar to operator[]

Returns the number of elements in the bucket with index bucketIdx.

If bucketIdx is not valid, the effect is undefined.

```cpp
#include <iostream>
#include <string>
#include <unordered_map>

int main ()
{
  std::unordered_map<std::string,std::string> mymap = {
    {"us","United States"},
    {"uk","United Kingdom"},
    {"fr","France"},
    {"de","Germany"}
  };

  unsigned nbuckets = mymap.bucket_count();

  std::cout << " mymap has " << nbuckets << " buckets:\n";
  for (unsigned i=0; i<nbuckets; ++i) {
    std::cout << " bucket #" << i << " has " << mymap.bucket_size(i) << " elements.\n";
  }
  return 0;
}
```

```
mymap has 5 buckets:
bucket #0 has 0 elements.
bucket #1 has 2 elements.
bucket #2 has 1 elements.
bucket #3 has 1 elements.
bucket #4 has 0 elements.
```

```cpp
  std::cout << " What does an invalid bucketIdx have?\n";
  std::cout << " bucket #" << (nbuckets) << " has " << mymap.bucket_size(nbuckets) << " elements.\n";
```

```
bash: line 7:  3634 Segmentation fault      (core dumped) ./a.out
```

What can we do?

What shall we do?

- Try-catch blocks?
  - No, there are no exceptions...
- Explicit checks?
  - Cannot protect against typos
- Reimplement the STL
  - Sometimes maybe...

Automate and Educate

- Listen to the compiler *(-Wall -Wextra -Wpedantic)*
- Use a sanitizer (g++/clang)
- Follow coding/naming guidelines
- Understand the concepts
- Practice contractual programming
- Share knowledge
  - Knowledge Sharing sessions
  - Code reviews

Conclusion

The STL is “full of” UB by design

Respect the contracts

Automation and education is key

Undefined behaviour in the Standard Template Library

Sandor DARGO

16th June 2020

C++ On Sea